TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

GRIGORIJ LJUBIN SAVESKI

ACCESSING NATURAL LANGUAGE PROCESSING ENGINES AND TASKS

Master of Science thesis

Examiner: prof. Mikko Tiusanen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on the 3rd of September 2014

ABSTRACT

GRIGORIJ LJUBIN SAVESKI: Accessing Natural Language Processing Engines and Tasks
Tampere University of technology
Master of Science Thesis, 48 pages
November 2014
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Mikko Tiusanen

Keywords: natural language processing, natural language processing tasks, natural language processing engines, part-of-speech tagging, interface

This thesis presents how a natural language task can be accessed through the use of natural language processing engine in an easy way. So far the access to the task of part-of-speech tagging and other tasks has been going through the engine command line interface, which demands both knowledge and experience in scripting and programming. Moreover, manual work had also been required to prepare the input data in order to be fed into the engine. At the same time all the output files from the task and engine have been handled manually.

To solve these issues, both the OpenNLP engine and its part-of-speech tagging task are integrated into a web interface that can be used by individuals that possess little or no technical knowledge. Furthermore, the system also guides the users through a process where they can input their data and it will automatically be processed and prepared for further use. After that they can follow the rest of the task and use the engine. At various points of the usage, the data is saved so that it can be used later to continue the process from wherever it was stopped. The data files are stored and organized on a server, which helps reusability. At the same time, the structure of the system is easy to extend with other language processing tasks and engines according to future needs. Last but not least, the current implementation makes the whole interface accessible from different locations and is quite portable. No graphical user interface details for the system will be presented in this thesis.

The resulting interface provides for ease of use, access, and expandability. Some challenges in the future include increased complexity of the system because of different tasks and engines. Moreover, certain parts of the process and the structure of the implementation could be improved.

PREFACE

I would like to thank my family for their support and the colleagues from Lionbridge for their valuable input on both the project and this thesis.

Tampere, 18.11.2014

Grigorij Ljubin Saveski

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Many of the methods that can be used to process natural languages can be accessed through the so called natural language processing engines. The problem is that for one to use many of the engines one needs at least some scripting and programming skills. Because of this, the obstacles for using the engines are increased substantially for some general users, for instance, linguists. And since it is challenging for such nontechnical individuals to interact with scripts or the command line interfaces of the engines, there is a need for software that follows one of the machine learning paradigms and allows natural language processing. Another issue is that the natural language data that is used with the engines usually needs to be processed manually by the users. This can be a hard and long process, especially, if one works with large amounts of data.

To solve those issues, an application was developed that targets nontechnical users. It allows the users to use the engines and tasks by following a flexible process that can be paused and continued. This can, of course, be achieved without any programming skills. The application also processes and stores the natural language data, used as an input, automatically.

The rest of this thesis has the following structure: there is an introductory chapter to the machine learning model that is used in the process of the application and a chapter on the details of natural language processing. The latter one explains the background on some of the machine learning and statistical approaches that will be used in the software. After that, there is a part that presents the OpenNLP engine [32], which uses the mentioned approaches, and how its tasks can be accessed. The next chapter shows how the process in the application is divided into different stages and what their inner workings are. The next part, presents one of the possible ways of how expandability of the application can be achieved, with various natural language processing tasks and engines. And last there are two chapters that evaluate various aspects of the interface and its current structure, and how some of the issues that are present now could be solved in the future.

2. ACCESS TO TASKS OF NATURAL LANGUAGE PROCESSING

Important parts of natural language processing are the so called natural language processing tasks that are approaches of solving some issues in the field. Various parts of natural language processing use those tasks to extract some meaning from a text or to handle it in different ways [26][35]. Some of the most commonly used tasks include part-of-speech tagging, tokenizing, parsing, name finding, and sentence splitting [1] [35]. The natural language processing engines contain the means of supporting the tasks [1][2]. The main focus of this thesis is the part-of-speech tagging task, how it can be solved and used through the OpenNLP engine [1].

The goal of this thesis is to enable the use and access to certain natural language processing engines and tasks without any technical prerequisite knowledge, such as script development and command-line interaction with different frameworks. The interface to the engines and tasks needs to allow *flexibility* in the workflow so that the users can access their previous uncompleted sessions and continue them without any difficulties. The whole interface needs to be easy to *expand* with any number of engines and tasks according to the needs of the users. Furthermore, it is important that the users are able to access the engines and tasks from *various locations* and the structure also needs to be *portable*. The goal of this thesis has been achieved with a web application, which makes the interface to the engines both portable and accessible. The main use of the application is the creation of models for various natural language processing tasks and their subsequent use in different fields of linguistics.

As far as natural language processing is concerned, part-of-speech tagging is one of the basic tasks. This task is often a prerequisite for further development or an improvement for other more complex algorithms and methods. A common problem statement that illustrates the need for low-cost part-of-speech tagging is the development of a morphologically complete dictionary for a language, e.g., for a spell-checker. In this case, part-of-speech tagging is necessary either for categorizing an existing corpus of words or for developing a morphological analysis tool to ensure completeness of the dictionary. A corpus is a large collection of textual data [22]. After this, more information can be obtained by observing the data, say, what are the most numerous parts-of-speech, find important words by tag (extract all the nouns or verbs), or just making the corpus more appropriate for linguistic research. [19]

2.1 Processes of creating task models

The tasks in natural language processing consist of several different stages. There are many different paradigms for this process, many of which are based on machine learning techniques, data mining, and pattern recognition. In a number of them [21][26][35], some repetitions in their stages can be seen, more specifically a training stage, which is then followed by testing stage. These and some other stages are explained further down. Additional stages can be added according to the requirements or if supplementary features from the process are needed.

Training a model for a task prepares it to make predictions based on that task. Afterwards, one can expect the model to behave as accurately as possible according to the information deduced from the training data, a corpus, for example. There are different forms of training, but here the concepts of supervised and unsupervised training will be discussed. These use annotated and unannotated data, resulting in supervised or unsupervised training, respectively [21][29]. In the case of part-of-speech tagging the annotated corpus would contain fully tagged information [1] and the unannotated would be regular text [29]. For other supervised tasks the data needs to be formatted according to the task or engine requirements.

Sometimes, the input training data is not organized in the way it needs to be or it contains some noise or unnecessary information, which needs to be filtered [13]. This preprocessing stage must be done before the training [13]. One could include instance selection into the preprocessing to handle some of the cases mentioned above. It is a technique, based on data mining, which can be used to lower the levels of noise and extract only the most crucial data from the input set [21]. This way the data will be ready for the training stage and there will be no mistakes or loss of data [13].

The testing stage is there to check how precise the model is and whether it conforms to the specifics of the task. There are different techniques used to evaluate the precision, which depend on the input data. For example, cross validation splits the training data into a large number of groups [21]. All of the groups are used to train the model, except for one, which is used to evaluate the model [21]. This process is repeated for every specific group and, at the end, the average of the testing scores represents the precision. The method to evaluate precision that is used here is to divide the input into two sets, training and testing data [19][21]. The proportion between these sets is usually predefined [21], but here there is some flexibility, since the users are allowed to choose the proportion between them. At evaluation, the accuracy of the model is calculated by dividing the number of correct predictions with the number of total predictions [1]. After this stage there are two outcomes, namely, one either continues with any other stage if

the results from the testing are satisfactory or goes back to the previous stages because of lack of precision of the model.

2.2 The employed process

In this thesis, a slightly modified version of the abovementioned process is used, which can be seen in Figure 1. The first stage is preprocessing where the users supply some input data, which is then filtered and prepared for the following stages. Then there is the training, where a part-of-speech tagger model is trained from a set of data. The testing of the model file finds out how it reacts to the data and what is the accuracy and consistency of its reactions. At the end of the process the users are able to use the model to fulfill the natural language processing task on whatever data they want. The last stage was added, since it is a relevant one for industrial use.

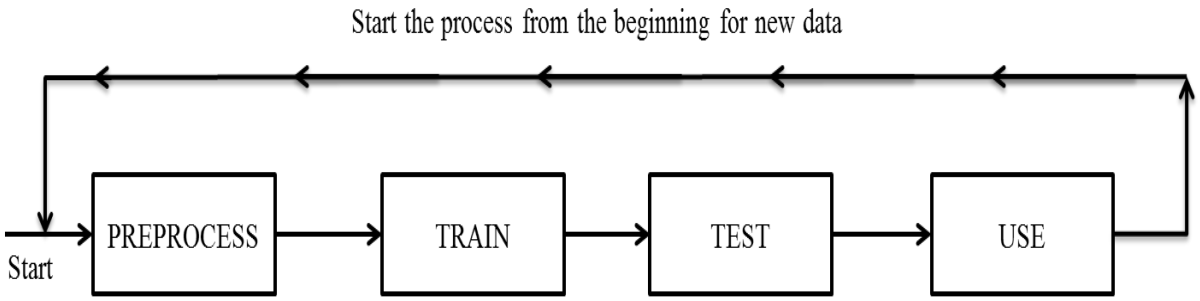

Figure 1. *Overview of the process*

2.3 Constraints

There were several constraints that were required by the company that financed this thesis. One of them was to use the Microsoft based ASP.NET framework. Moreover, the code behind had to be developed in C#. Other two tightly connected constraints were to at least implement the OpenNLP engine into the application and to have an expandable interface to the tasks and engines.

3. NATURAL LANGUAGE PROCESSING

Natural language processing is an area of computer science which is tightly connected with many disciplines such as linguistics, machine learning, artificial intelligence, statistics, and information theory, all of which belong to different fields. The main point towards which it strives is making machines understand and analyse natural languages, in either textual or verbal form, as well as or maybe even better than any human can.

Natural languages are one of the forms of communication between humans. The problem is that humans do not always speak (or write) plainly, which is why it is not easy for a machine to understand certain parts of speech and text. Moreover in many languages there are words that have more than one meaning. And hence many problems that arise in natural language processing come from the many levels of ambiguity present in languages [26].

Humans analyse text and speech from different linguistic points of view when dealing with languages. When using the multiple levels of processing (syntactical, semantic, phonological, or lexical) they gain an insight on the numerous meanings of what is written or spoken, the rules that are applied or even what the context of the data is [6][22]. According to Liddy [22], in order to make a humanlike system for processing natural languages (which so far has not been achieved) one must make a system that uses the different linguistic levels as humans do. Moreover, Liddy makes a difference between natural language understanding and natural language processing. The understanding comes when a machine is capable of several different outcomes when it is given some text [22]. First of all the system should be able to paraphrase the data. Second, it should translate the text between different languages. Thirdly, it should be able to answer questions based on the input. And finally it should draw conclusions from what it understood. Natural language processing is actually striving towards natural language understanding as its final goal, when a machine will be capable of humanlike level of processing when dealing with languages [22]. The foundations of natural language processing can be applied to large number of areas such as speech recognition, machine translation, artificial intelligence, and text processing. [6][22]

3.1 Short history

Natural language processing started in the late 1940s with the primary area of interest being automated translation. Most of it was based on the foundations laid out or connected with cryptography and information theory [22], which were increasingly researched since the beginning of World War II. This was a period plagued by low computational power and storage of the computers [18]. Moreover, much of theoretical basis was not developed at all. The translation process was done by converting the words to other languages and reordering them so the rules of the output language are preserved [18][22]. Hence ambiguity caused many of the problems in natural language processing and it still cannot be fully resolved today. Many of the systems of the later period dealt with the field of artificial intelligence, while much of the linguistic theory was not used at all [18][22].

The 1970's and 80's were the periods where some of the techniques that were used in natural language processing included language generation. This was mostly done by predicting an output text based on the input and by drawing conclusions based on the supplied text. With the growing power of the technology used, more and more progress in the field was also achieved. At this time the use of statistics and probability was on a rise since it was noticed that these offered methods of accomplishing some of the goals of the field. Another approach that was established at that time was the use of large sets of data to train and test the systems in machine translation, something which is still used nowadays. [18][22]

In the 1990's several factors contributed to the growth of natural language processing. One of those was the improvement of hardware which led to better computers with greater processing and storage power [18]. Another factor was the concentration on smaller natural language processing tasks, a kind of divide-and-conquer approach, and the applications of generalisation and abstraction on the data and the tasks [18][22]. Also, more and more data was freely available for use on the internet along with the creation of new corpora and expanding the old [18][22]. Moreover, more systems were made that autonomously handled and extracted the needed data. With the use of statistical methods it was possible to approach many of the issues met in linguistics like part-of-speech, word extraction, and word frequency [18][22]. Moreover, since precision and correctness were very important, and still are today, evaluation of the performance of different natural language processing tasks was also being developed further [18].

After the year 2000, there has been even further progress in the field of natural language processing with the use of better algorithms and methods of handling input data, computers with even better performance than before, and huge amounts of data [18][22]. Additionally, more advanced systems and research foundations have also brought better results than in the past. These reasons are behind the progress in, say, machine translation or information retrieval by the search engines, which are able to deal with massive

amounts of textual data [5][18]. Despite these we are still discussing mere natural language processing and not natural language understanding, which is still the ultimate goal of this discipline [5][18][22].

3.2 Task examples and issues

Let us consider some examples of natural language processing tasks. Tokenization is the process of splitting some text into its atomic units, tokens. They mainly include all the words that comprise a sentence, although punctuation marks and the rest of the symbols are also considered [10][43]. Text parsing, or just parsing, is the task of identifying groups of words in a sentence, which are connected through the grammatical structure of the sentence [20][40].

In natural language processing part-of-speech tagging is the process of marking each part of the sentence with its part-of-speech [19][42]. Part-of-speech contains a number of categories (nouns, verbs, adverbs, etc.) which can be used to label different words [19][42]. Through those categories we can learn more about the words themselves and their neighbours. For instance, if we consider the words “bank” and “go” in the sentence “I will back you up when you go to the bank.”, we know right away that they are a noun and a verb respectively. With the help of part-of-speech a difference can be made between various kinds of words and other parts of the sentence, which in turn can give a lot of information about the words that precede or follow [19][42]. That data can later be used in other tasks [1].

However, as was already mentioned, a big problem in part-of-speech tagging is the ambiguity of words, that is, the tag depends largely on the context of the words. In the example: “I will back you up when you go to the bank.” a closer look will reveal that some words can be tagged in multiple ways or have more than one meaning. For instance, the part-of-speech for the word “will” can be either a modal verb or a noun, “back” can be a verb or a noun, and “up” can be tagged as a particle or a preposition. Moreover, what is the actual meaning of “bank”? It is obviously a noun and will be tagged as one but it is unavoidable that there is ambiguity. Is it meant as a river bank or a building where financial matters are handled? It all depends on the context which can be modified by the preceding or following words and even sentences. When a person reads the sentence and understands what each of the words mean (depending on the context of their use) they can get the difference between “back” as a verb or a noun and the river bank or the other bank, whereas a machine could not because of the multiple interpretations. Some words (like “bank”) can be disambiguated by an automaton using the tags of preceding words (“the”, which implies a noun). But there are others (like “back”) that are not so easy to disambiguate since the preceding words are also ambiguous (“will”). Furthermore, there are even other cases where the context is complex and

not as clear as in this example. In cases like that, ambiguity can only be resolved with semantic knowledge about the whole text [22]. This means that the machine needs to understand what is conveyed through the text, like a human would know. The difficultly is that no such systems have yet been invented [22]. The above mentioned issues are some of the problems that are faced in part-of-speech tagging. [19][26][42]

3.3 Approaches for using natural language processing

Although there are several different approaches [22] to handle natural language processing, its tasks, and its issues, we will focus on the statistical (or probabilistic) method since it is the one which is used for the part-of-speech tagging that is part of the following chapters. In it, the model for the task is trained on substantial corpora and it learns statistically about the rules of the task [3]. Hence, through the probabilistic approach the model learns without any linguistic knowledge [3][26].

Furthermore, there are many statistical methods for using natural language processing [22] but, here, only two closely related ones will be considered. The first is based on Bayes' theorem [25][37] and the second is based on machine learning techniques. As stated by Ratnaparkhi [35] and Marquez [26], the Bayesian methods make independence assumptions that are learnt from features that come from whatever training corpus is used. In particular, features [25][35] are considered to be binary or Boolean functions and hence return one or null (true or false) depending on the case. So, using those functions all the probabilities are attained from the data.

The second method is based on maximum entropy. According to Ratnaparkhi, the maximum entropy framework makes no independence assumptions although it still uses corpus data to learn [35], similarly to the previous method. It makes use of algorithms, through which features, as above, are formed. In this framework, decision trees keep all the knowledge of the model, upon which probabilistic choices on how to handle the data are made. The creation of the decision trees is based on rules [25], which will be discussed later. The machine learning techniques applied are discussed in further detail in the following chapters, after a closer look at maximum entropy is taken. [3]

3.4 Maximum entropy

The principle of maximum entropy is very simple in nature. It says that when faced with two (or more) choices for which it is not clear which is more possible to happen, one should consider that all the choices have the same probability (uniform distribution)

[3][25]. Its main principles, in one way or another, are known to date back to Herodotus (fifth century BC) and his work [3].

Moreover, a parallel can also be made to Occam's razor, which was first stated by William of Ockham (or Occam) [3]. It is a principle where the simplest solution that solves the given problem should be chosen. The original form of the razor is: "Pluralitas non est ponenda sine necessitate", and one of the many translations is: "entities should not be multiplied beyond necessity". [28]

Maximum entropy was initially developed for use in statistical physics. In this field the concept involves finding out the probability distributions of the elements inside a defined system, without presuming more information than what is available at hand [17]. Since its conception, maximum entropy has found numerous uses in different fields, for example, natural language processing [4][23], biology [34], economics [14], and information theory [39].

3.5 Maximum entropy for natural language processing

According to Manning and Schuetze [25]: "Maximum entropy modeling is a framework for integrating information from many heterogeneous information sources for classification". The heterogeneous information sources may comprise many samples. In the example of the part-of-speech tagger that was used previously, the samples can be, for instance, all the smaller parts of the sentence that contain the word that is supposed to be tagged. The samples are important because they are used in the training of the model to teach it the different rules and features. Both the rules and features are created from constraints that are learnt from the training data. Through all of them the distributions of the probabilities of different outcomes can be calculated within the framework. At the end, the outcomes that are chosen are the ones that satisfy the rules and features, and through them the constraints [35]. They are the ones that conform to the maximum entropy distribution, that is, that have highest entropy in the probabilities. [3][25]

Let us try to apply the principle of maximum entropy to a task in natural language processing similarly to Berger et al. [3]. First of all, some large input of data, a corpus, needs to be assembled that will be used to train a model that will do the task. Moreover, let us imagine that there is some word in a certain language, which can have three distinct tags attached to it according to the input data. Let us imagine the word can be tagged as a noun, a verb or an adjective. This will be the first rule (or constraint) upon which the model is trained to behave according to, when it encounters the word. Because of that, it has a very uniform distribution since it allows the same chance to all of the three tags. So, all the tags have a 33% chance to be chosen. Let us imagine another fact, which can later be noticed from another piece of the training data: noun and adjec-

tive are preferred most of the time. Because of this second rule the probabilities of those two tags go up. Once the training ends, it can be noticed that the model is trained on just two rules, for the word that we are considering. Because of the fact that the part-of-speech tagger was created on the principle of maximum entropy, whenever it will encounter the word in some test data, the model will take into account the probabilities from the two rules it was trained to use and nothing else. The word will be marked with one of the tags that satisfies the rules, and at the same time is as uniform as possible [35]. [3]

But these kinds of statistical rules are not the only ones considered in maximum entropy models for natural language processing. Many of them would be tied to various contexts such as the order of the words in the sentence or previously assigned tags. Contexts can also be used for various other tasks of natural language processing, not just part-of-speech tagging. [3][35]

For contexts, let us consider: “I will back you up.” and “He was shot in the back.” The word “back” in these two sentences can be tagged with two different tags, verb or a noun. So, if a maximum entropy tagger considers the words around “back” it might notice that in the first sentence “back” is preceded by “will” and followed by “you”. If it is trained on some data that had similar format then the tag verb would have higher possibility than the others. Or in the second sentence the tagger will notice that the word is preceded by “the” which would mean that there is a high likelihood that “back” is a noun. [35]

The above shown examples can be considered a feature (or a feature function) in Ratnaparkhi’s approach [35] for handling of natural language processing tasks. They are based on the features from the maximum entropy framework. If we use one of the above examples for part-of-speech, a feature function would be: if the word that is considered is “back” and it is preceded by “the”, return true, otherwise return false. The part of the feature that checks if the preceding word is “the” is known as contextual predicate. Further concrete examples of a contextual predicates that are used in the approach are “the word contains uppercase character” and “the word contains a hyphen”. As it can be observed they are also Boolean in nature. Of course, feature functions and contextual predicates can be created for other natural language tasks, not just for part-of-speech. The feature functions, for any task, are chosen to be included in the model during its training only if they have been seen at least ten times. In the testing stage, the features along with the statistical rules can affect how some words are tagged with their part-of-speech. [35]

However, part-of-speech tagging is only one of the natural language processing problems that can be at least partially solved with the application of machine learning techniques. While the application that is discussed here is focused on solving the part-of-speech tagging problem by using maximum entropy from OpenNLP, the infrastructure

that is created in the process can also act as the blueprint for other machine learning solutions and natural language processing task, such as tokenization and text parsing. [19]

3.6 Part-of-speech tagging based on maximum entropy

Part-of-speech taggers contain two main parts: a model and an algorithm, both of which are closely associated [26][35]. Models are created when the tagger is fed input data in the training stage. Then, when the tagger is used during testing, the algorithm draws conclusions on how to do the tagging based on what the model has learnt [35]. Since these two parts are so closely connected and embedded into the tagger together we sometimes use the words model and (part-of-speech) tagger interchangeably. There are many algorithms on how to apply part-of-speech tags to words in a text. For instance one can use Hidden Markov Model tagging, memory based tagger, rule-based tagging, or maximum entropy tagging [8][19][26][33][42].

Let us focus on the maximum entropy tagging, since the OpenNLP tagger is based on its principles [15][27]. Because of that, almost all of the methods used in the tagger are based on Ratnaparkhi's [35] maximum entropy tagging. Hence, OpenNLP makes use of various rules, features, and contextual predicates that govern the probabilities of different words and they are created and chosen during the training stage of the tagger. Like it was already mentioned, the number of times a feature is seen is very important for the model, since if it appears rarely in the data it may bring inconsistent probabilities and hence the model might not be able to predict it well [35]. That is why limits are placed on the minimum number of times that they must be seen or they are discarded [35]. OpenNLP leaves it to the user to decide what would be the minimum, known as cutoff num for part-of-speech, but does not enforce any kind of restrictions on this [1].

Once the training of the tagger is done, the next stage would be to test it out on some data and see how it behaves. If given some text, it works on this sentence by sentence. While it goes through a sentence the tagger creates several different probability structures on how to tag each word in it. The structures are based on the features and rules. At the end, the structure with the highest probability, for the sentence, is chosen as the most suitable one and the words are tagged accordingly. Moreover, in order to increase the correctness of the tags the maximum entropy tagging algorithm uses a so called tag dictionary. It contains all the possible tags for each word it has seen in the training stage. Let us imagine that the tagger has seen the word "plant" while in training and its tags in the dictionary are "noun" and "verb". So, when it encounters "plant" in testing, the only tags that will be contemplated for that specific word are "noun" and "verb". If the word that is considered has not been seen previously, then the tagger will consider

all the possible tags for it. OpenNLP also allows the use of tag dictionaries in its work and they are implemented on the same principle [1]. [35]

Part-of-speech taggers based on Ratnaparkhi's work have been proven to show more than 96.5% precision in their work [35]. There are some that even have shown further increase over the past years, albeit they have only brought around another per cent on top of the past results [24]. Despite these percentages being quite high, they do not realistically represent the actual outcomes from some cases of the use of part-of-speech taggers. That 97 % is the outcome when tagging each token of a text separately [24]. When whole sentences are considered the percentages fall to those around the fifties [24]. This is mainly because of the ambiguity that was already discussed and the fact that an incorrect tag in one sentence may bring an avalanche of mistakes in the same or in the following sentences [24][26]. So, even though some part-of-speech taggers are now more powerful than humans at doing the task, they are still far from perfect [26][35].

4. OPENNLP

OpenNLP is a natural language processing engine based on machine learning and it was created to satisfy the need for high quality framework for the purpose of language processing [1]. It allows its users to create models for multiple natural language processing tasks [1]. OpenNLP makes use of the maximum entropy framework for some of its tasks [27]. [2][32]

OpenNLP was created in the year 2000 by Jason Baldridge and Gann Bierner. According to one of the web pages about the engine [32]: "OpenNLP, broadly speaking, was meant to be a high-level organizational unit for various open source software packages for natural language processing; more practically, it provided a high-level package name for various Java packages of the form opennlp.*". It started out as the Grok toolkit [32] for natural language processing and offered an interface to the tasks as an OpenNLP API (application programming interface) [1][2]. Grok was the one that contained the processing functionality and offered the tasks. [32]

In 2003 the Grok toolkit and OpenNLP became two distinct entities: Grok became OpenCCG [32] and henceforth OpenNLP became to exist as its own toolkit, because the creators wanted to make a distinction between the two separate functionalities. From there on, OpenNLP contains the APIs and the natural language processing functionality from Grok. Since then they have had their own development processes. [32]

OpenNLP supports many natural language processing tasks: part-of-speech tagging, chunking, coreference resolution, parsing, named entity extraction, sentence segmentation, and tokenization, all of which will be explained in this chapter. With these tasks, one could build more complex systems, if required. All of the above mentioned modules include both APIs and command line interfaces, through which it is possible to train and, if required, to evaluate the tasks. OpenNLP contains a large number of auxiliary packages, some of which are specific to certain languages, like English or Spanish, while others can store language rules. These are only limited to a handful of natural languages. Other packages can be used to convert a large number of corpora to a format offered by OpenNLP. All the examples used in this chapter are taken from the official documentation [1] about the engine. [1][2]

4.1 Sentence detector

The OpenNLP sentence detector allows the users to identify sentences in a text, that is, to find the punctuation mark that ends each one of them and modify the text by putting each sentence on a separate line. Maximum entropy is used to identify if different punctuation marks are the actual sentence finishers. Hence, the OpenNLP sentence detector does not distinguish sentences depending on their contents: everything is based on rules. [1]

If one wants to make an OpenNLP sentence detector, there are several classes in its package, the most important one being the class SentenceDetectorME. This contains the main functionality for creating a sentence detector model based on maximum entropy. Hence, one could use the method train to make a detector for this task. In order to do this, the input text needs to be passed as a stream to the function and the factory and training parameters set. The factory includes a lot of methods to help the creation of the model and extend its functionality, for instance, to create a map for organization of the data, find the ends of the tokens, and many getters to return various fields. The training parameters, on the other hand, define which algorithm will be used in the training process, how to work with the map created from the factory, and how to serialize the model. [2]

Besides the training method, the SentenceDetectorME class also contains a number of helpful auxiliary functions. One of them is getSentenceProbabilities which returns the probabilities of the previous calls to the sentence detector. Other examples are sentDetect, which splits a string of text into sentences, and sentPosDetect, which can find the first words of sentences. Of these, sentDetect is especially important, since it is the function invoked by the model, after its training, to implement the natural language processing task on an input string. [2]

To ease and to improve the work of SentenceDetectorME, some additional classes can be used. The class SentenceSample has methods to retrieve documents and sentences and to find their starting indexes. On the other hand, SentenceSampleStream makes the preparations for the sentences for the previous class. This is done by reading and then filtering the samples of text and converting them to objects. One could also use the class SDCrossValidator to cross validate the results of the sentence detector. Another evaluator that can be used is the SentenceDetectorEvaluator which, through its method getFMeasure, can calculate the precision of the sentence detector model. There is also the SentenceModel class used to encapsulate the models and to write the model file. [2]

Let us briefly consider an example of this task to see how it actually works. If for instance the following text is given as an input to a model:

Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29. Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group. Rudolph Agnew, 55 years old and former chairman of Consolidated Gold Fields PLC, was named a director of this British industrial conglomerate.

the output of the task would be:

Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29.

Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.

Rudolph Agnew, 55 years old and former chairman of Consolidated Gold Fields PLC,

was named a director of this British industrial conglomerate.”.

One should note that each sentence in the output is on a new line. [1]

4.2 Tokenizer

The OpenNLP tokenizer, as the name suggest, splits whatever text is given to it into tokens. The tokens here include words, punctuation marks, and numbers. It first divides the text into sentences (using the sentence detector) which are then tokenized. The OpenNLP tokenizer has three versions, whitespace, simple, and learnable tokenizer. One can also use a detokenizer to return the data to its initial format. [1]

The main class to train a maximum entropy model for the tokenization task is the TokenizerME. It, of course, contains a method for training that needs an instance of TokenizerFactory, to set the resources, and TrainingParameters, to regulate the settings of the tokenizer. In short, it is very much like the training method for the previous task from OpenNLP. Some additional functions of the class are tokenize and tokenizePOS. The first one splits whatever input string is given to it into tokens and hence contains the main functionality for this task. The second function finds where the tokens start and end. The probabilities of the previous uses of this class can be accessed through the method getTokenProbabilities. [2]

There are other useful classes that can be used for achieving greater flexibility with this task or can be used in conjunction with the TokenizerME. For instance, there are the SimpleTokenizer and WhitespaceTokenizer, that both contain instances of the methods tokenize and tokenizePOS. Other examples include the pair DetokenizationDictionary and DictionaryDetokenizer that can do the reverse, detokenization. The rest are auxiliary classes that are used for streaming data from files, evaluation the models in different ways, or for creating dictionaries. [2]

Let us consider an example where the whitespace tokenizer is given the following input:

Rudolph Agnew, 55 years old and former chairman of Consolidated Gold Fields PLC, was named a director of this British industrial conglomerate.

The output would then be:

Rudolph Agnew , 55 years old and former chairman of Consolidated Gold Fields PLC , was named a director of this British industrial conglomerate .

Note that every token (even the punctuation marks) are followed by whitespace. [1]

The above mentioned tasks can be very useful since many of the other tasks supported by OpenNLP need the input formatted in this way. Moreover, the tokenizer can be used even if one does not plan to otherwise use the engine, for the simple reason that it is a way of preprocessing the data that makes its automated analysis and processing much easier. [1]

4.3 Name finder

The OpenNLP name finder is able to identify numbers and names in a string. Before this task can be used, a model for it needs to be trained using some corpora so that it can distinguish names from different languages. Moreover, in order for it to work, the two previously mentioned tasks should be performed first. [1]

The main class for the OpenNLP name finder is called NameFinderME. It also includes a method train, like all the others, but this one requires different parameters. One of them is an AdaptiveFeatureGenerator that creates a ruleset of features for the identification of names. The others are the resources for the task, the number of iterations that the function will make, and the cutoff. [2]

NameFinderME contains some other methods. For instance, find creates the name tags for a string input and identifies each name with its tag. This is the function that does the task over a given input. The method clearAdaptiveData, on the other hand, deletes the data gathered from all the previous calls to find and is useful at the end of several sequences of data. The class method probs returns the probabilities that were calculated for the last use of the name finder. [2]

Some of the other classes for the OpenNLP name finder differ from those of the other tasks (evaluators, stream readers, and cross validators) like NameSample, which contains methods to parse data, extract and store sentences and names, and to get various contexts needed for the maximum entropy. Class RegexNameFinder represents a name

finder that bases its rules on sequences of regular expressions. It contains its own find and clearAdaptiveData methods. [2]

Let us consider an example of the use of the name finder. If a model of this task is given the following text as input:

Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 . Mr . Vinken is chairman of Elsevier N.V. , the Dutch publishing group .

then the output would be:

<START:person> Pierre Vinken <END> , 61 years old , will join the board as a nonexecutive director Nov. 29 . Mr . <START:person> Vinken <END> is chairman of Elsevier N.V. , the Dutch publishing group . [1]

4.4 Document categorizer

OpenNLP document categorizer orders input data into different groups that need to be predefined by the users. A maximum entropy model is required to be trained on some data to use the categorizer. The input needs to be divided into the groups that will later be used to classify whatever text is given to the model. [1]

The OpenNLP document categorizer package, besides the various evaluators, stream readers, model creators, and sample holders, contains the class DocumentCategorizerME. This class has two important methods: train and categorize. The first one is similar to the training function for the name finder and takes the same parameters. The second method does the main functionality, namely, categorizes any given text. Class DocumentCategorizerME handles the different categories and results from the input texts. For instance, there are methods to return all, some or just the best of the groups of results. Their places or their number can also be extracted from the data. The unique classes include the BagOfWordsFeatureGenerator and the NGramFeatureGenerator. Both of them generate features for the words in a document based on their own principles. [2]

Let us consider an example for the document categorizer from OpenNLP, based on a Gross Margin category. If the input is the following sentence:

Major acquisitions that have a lower gross margin than the existing network also had a negative impact on the overall gross margin, but it should improve following the implementation of its integration strategies.

then it would be put into the category for decreasing gross margin, here named GMDecrease. On the other hand, if the input is:

The upward movement of gross margin resulted from amounts pursuant to adjustments to obligations towards dealers.

then it would be classified as GMIncrease. [1]

4.5 Part-of-speech tagger

The OpenNLP part-of-speech tagger goes through the input text token-by-token and predicting a tag for each one of them based on the maximum entropy part-of-speech tagging [15]. Again, it is important to note that the probability of a tag over another depends on the token in question and its context. Any tag dictionaries are purely optional and need to be provided by the user. Their use can speed up the algorithm and it can also lower the number of incorrectly assigned tags for each token. [1]

The OpenNLP part-of-speech tagger uses the Penn Treebank set of tags [1] to mark the tokens. It is one of the ways used to tag that the words are nouns, verbs, pronouns, etc. [36]. A model needs to be trained; the training input needs to be properly tokenized, annotated, and formatted, namely, it should contain tokens along with their tags and one sentence per line. The format of the tokens required here is token_tag. It is, of course, very important that all the tags assigned in the training data are correct. A separate model needs to be created for every language and appropriate data in the same language needs to be used. [1]

The class POSTaggerME methods return the number of predicted tags, order them, or get the probabilities for every tag in a sentence. The method tag, which has several different instances to handle various types of data, performs the tagging on any input that is passed. Some methods create part-of-speech dictionaries that can be used in the task, such as buildNGramDictionary and populatePOSDictionary, based on their own principles. The POSTaggerME also contains a method for training a model and its practical use can be seen in the following chapter. [2]

The OpenNLP part-of-speech task also has some exclusive classes. Class POSSampleEventStream has methods for reading objects from the class POSSample. Then they can be turned into events and, later, used by the maximum entropy library in the training process. [2]

Class POSDictionary can be used to read tag dictionaries and find out which tags go with each word. Its method getTags can be used to return all the tags for a certain word,

while put can associate a list of tags with a word. The other methods can be used to extract the data from the dictionaries in various ways. [2]

The class DefaultPOSContextGenerator, on the other hand, can through its methods produce context for each token that is passed to it. The context is created from the relation between each separate token and every tag that has been assigned to it in the past. This is one of the pieces essential for the maximum entropy framework. [2]

The OpenNLP part-of-speech task also has an evaluator class, called POSEvaluator, which is different from the evaluators for the other classes. Its methods can be used to get the number of correctly identified tags and the total number of words that were taken into consideration. This can then be used to calculate the precisions of various taggers created with the other classes. [2]

An example of the use of the OpenNLP part-of-speech tagger is the following: the input sentence is

Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .

while the output would be:

Pierre_NNP Vinken_NNP ,_, 61_CD years_NNS old_JJ ,_, will_MD join_VB the_DT board_NN as_IN a_DT nonexecutive_JJ director_NN Nov._NNP 29_CD ._. [1]

4.6 Chunker

OpenNLP also allows the use of a chunker. Its function is to organize the various syntactical elements of the input text into groups. The chunker first uses a part-of-speech tagger to tag the words of a sentence after which they are split into syntactic groups, like verbs, prepositions and particles. Of course, the data needs to be properly formatted, and in this case every word is required to be in a new line. The word is followed by two tags: the first is its part-of-speech tag and the second is a chunk tag. [1]

The OpenNLP chunker again has classes, similar to those of others: the class ChunkerME has methods to train, use it on various types of data, or compute the precision of previous chunking. Its unique method can return a list of chunks for a sentence. [2]

Consider the following (tagged) sentence:

Rockwell_NNP International_NNP Corp._NNP 's_POS Tulsa_NNP unit_NN said_VBD it_PRP signed_VBD a_DT tentative_JJ agreement_NN extending_VBG

its_PRP$ contract_NN with_IN Boeing_NNP Co._NNP to_TO provide_VB structural_JJ parts_NNS for_IN Boeing_NNP 's_POS 747_CD jetliners_NNS ._.

Then the output of the task would be this:

[NP Rockwell_NNP International_NNP Corp._NNP] [NP 's_POS Tulsa_NNP unit_NN] [VP said_VBD] [NP it_PRP] [VP signed_VBD] [NP a_DT tentative_JJ agreement_NN] [VP extending_VBG] [NP its_PRP$ contract_NN] [PP with_IN] [NP Boeing_NNP Co._NNP] [VP to_TO provide_VB] [NP structural_JJ parts_NNS] [PP for_IN] [NP Boeing_NNP] [NP 's_POS 747_CD jetliners_NNS] ._.

As it can be noticed the tokens, along with their tags, in the output are grouped using square brackets. [1]

4.7 Parser

OpenNLP contains a parser which divides the input text into tokens which are then grouped according to their syntactical relation. At the end of the process, one can also choose to print the parse tree on screen if it is needed. When the parser is trained, a part-of-speech tagger will also be created at the same time so that the text can be parsed and tagged at the same time. For more precision this default tagger can be replaced with one trained by the user, by using its API, by using the classes from the section about part-of-speech for OpenNLP. [1]

The OpenNLP parser has classes and structures for storing parse constituents and retrieving various data about them. The class Parse contains methods to handle nodes in the parsing sequences, nodes can be added and removed, relations between the nodes can be changed, or parts of the structure can be cloned. There are also several functions that return different probabilities for the parsing sequences. The method show and others like it are used for visualizing the results of the parsing by the essential method parseParse. The unique class in the package is Cons, which has methods for storing and retrieving the features of the different nodes. [2]

If the following input sentence:

The quick brown fox jumps over the lazy dog .

is used the output from the model would be:

(TOP (NP (NP (DT The) (JJ quick) (JJ brown) (NN fox) (NNS jumps)) (PP (IN over) (NP (DT the) (JJ lazy) (NN dog))) (. .))). [1]

The words in the sentence are grouped in a parse structure if they relate to each other on a syntactical level. Because of that: The, quick, brown, fox, and jumps are in one group, while the, lazy, and dog in another. At the same time all those words also belong to the sentence and are in another group for the whole sentence. The tags before the words are parts-of-speech. [1]

4.8 Coreference resolution

The task called coreference resolution links multiple mentions of an entity in a document together [1]. The OpenNLP implementation is currently limited to noun phrase mentions only [1]. This has four packages:

coref, with classes to train a model, create discourse models, handle discourse entities and elements, create coreference resolution for Treebank parsers, and read/write of various types of data; [2]

mention, with classes to control, generate, make context for, and find mentions, or to handle dictionaries and to parse the data; [2]

resolver, with classes for resolution approaches needed, some employing maximum entropy, by identifying proper nouns, plural and singular nouns, pronouns, and appositives; [2]

sim, to identify the similarity between mentions with classes to enumerate, model, and store genders, numbers, and semantic types, class Context to create context for mentions based on the above types, and MaxentCompatibilityModel to create a maximum entropy models for mentions. [2]

5. CREATING A PART-OF-SPEECH TAGGER WITH OPENNLP

The goal of the thesis is to make matters easier for the users when interacting with natural language processing engines, like OpenNLP, and to allow them to use the different tasks, like part-of-speech tagging, that are supported by the engines. The users need to supply some input, which is then used by the engine to make a model for a specific language and task. The parts of the application that interact with the user data are shown in Figure 2 as stages. Most of them are based on the approach of creating natural language processing models, which was shown earlier. Moreover, since OpenNLP only supports this, the supervised method of training is used for the part-of-speech task, not the unsupervised one [1].

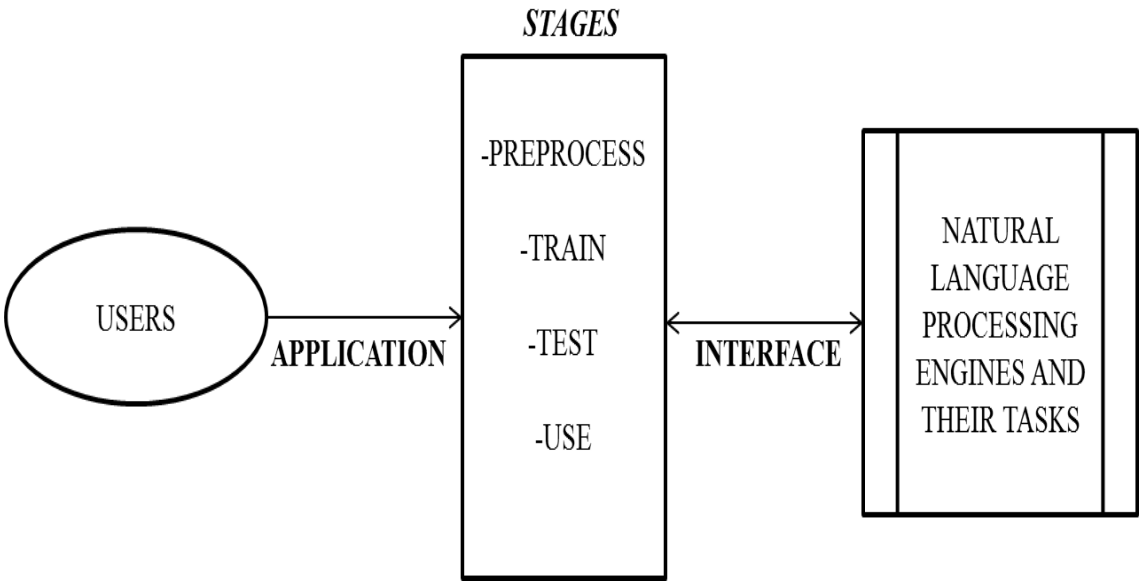

Figure 2. *The interaction between all the elements in the environment*

The stages are the ones that verify and handle the information in several ways to make sure that the engine receives what it needs. The input data needs to be properly formatted and the words need to be correctly tagged with their part-of-speech.

The OpenNLP part-of-speech tagging task was the only one implemented, even though the sentence detector and the tokenizer are the base for the other more complex tasks. This was because all the data that was provided for the tagging was properly formatted. Also one should note that the API, and not the command line interface for the part-of-speech tagging, was used since the intention was to use the tagger in a web application. [1]

5.1 Flexibility in the process

The part-of-speech tagging process (Figure 3) in the application is to some extent straightforward. If the users only use the application to preprocess their data then to train a model from it, test the model, and, at the end, use it, then the process is direct. But this is not true if the users had to stop in the middle due to some unforeseen event. That is why the users need some flexibility, that is, they need to have multiple entry and exit points in the flow. So the users can start or pause at any of the stages and continue to the following ones.

One example scenario where the pausing could prove useful would be if a model was trained and tested, but it showed poor results. Then this process can be paused by jumping to the training stage, where an older model is loaded, and it is tested with the data from the paused process. If the results are satisfactory the process can continue to the use a model stage. The entering and continuing is handled by the users through the user interface of the application.

The problem that is usually faced with these kinds of flexible processes is to balance the freedom of the user with some restrictions. The latter are needed so that the code knows where the user entered or exited and what conditions were met when the action happened.

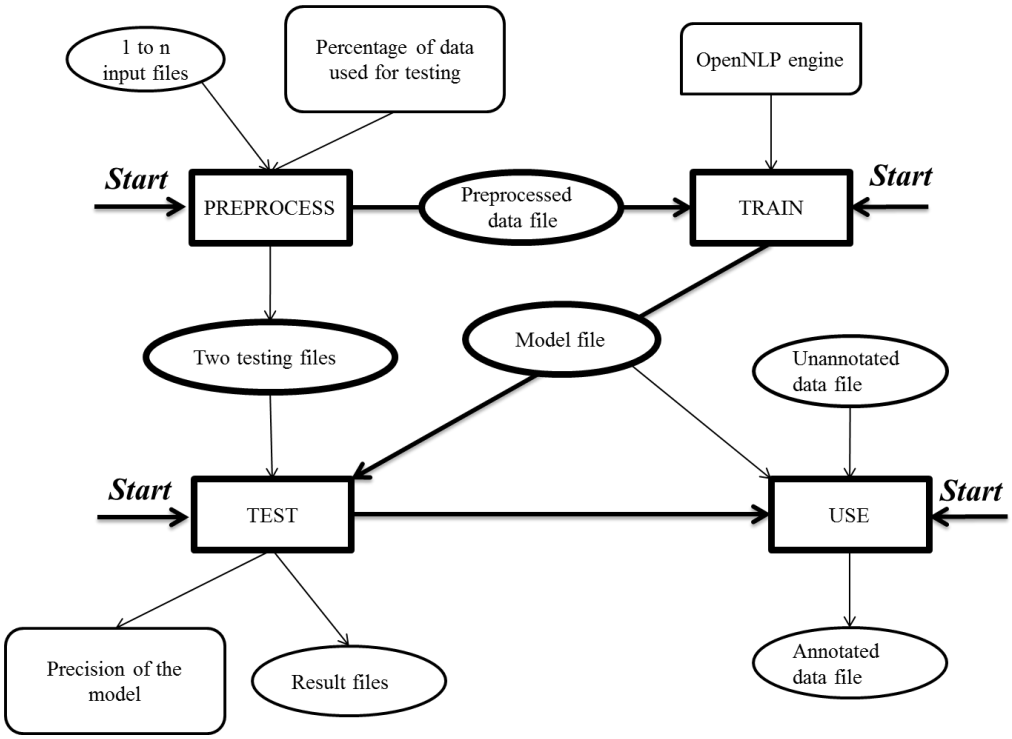

Figure 3. *The stages of the process with the inputs and outputs*

The users were restricted by making the input files, for all the stages, requirements to start them. So, to enter the training stage the preprocessed data file must first be provided, if one does not come directly from preprocessing. Moreover, the application follows the progress of the users by marking which stages of the process were already completed. This was done by saving the physical address of some of the output files, which are used as input in later stages. Those files include: the preprocessed data file, the two testing files, and the model file (marked with bold outline in Figure 3). Moreover, every output file can only come from a certain stage of the process, which means that every one of them is unique, hence making the processes of marking and checking the entry requirements easy.

Whenever the users enter a stage of the process there are two possibilities: they either came from the previous stage or jumped directly to where they are now. In the first case, they have already satisfied all the requirements and can freely use the functionality of the stage. In the other, the users have skipped at least one of the stages in the processes. Because of that, they are missing one or more input files and they do not to satisfy all the requirements in the stage where they end up. This is not a big problem, though, since they can freely load files from the server or from their local machine into the software to fulfill the entry requirements.

5.2 Server file selection and directory structure population

To allow file selection from the server, a tree structure was developed that will list the required folders that are present on the server. The folders are organized in four levels: username folders, language folders, task folders, and directories that hold all the files. The username folders are needed because each user should be able access all the data of other users. Language and task directories are needed because the number of languages and tasks will be greater than one. The last set of folders is there for better organization and easier access of input and output files for the code. It contains a directory for every stage in the process. Also all the four levels of directories are, of course, created for each user and each subsequent level is contained within the previous one.

There are several issues that need to be considered and solved with the tree. First of all, its algorithm should not list the fourth level of directories or any of the files inside since this will overcrowd the tree and cause confusion. Merely selecting the task directory should be enough to fulfill the entry requirements of any stage. Another problem was that some of the files may not exist because of the flexible nature of the whole process. This means that there may be empty folder branches which should be filtered out. For example, if user2 (in Figure 4) never managed to create a model file then his/her branch for Indonesian should not be included in the tree. On the other hand, user3 has model file for Indonesian so that branch will be shown. Furthermore, the tree should only contain the folders that have files which are connected with the particular stage of the process.

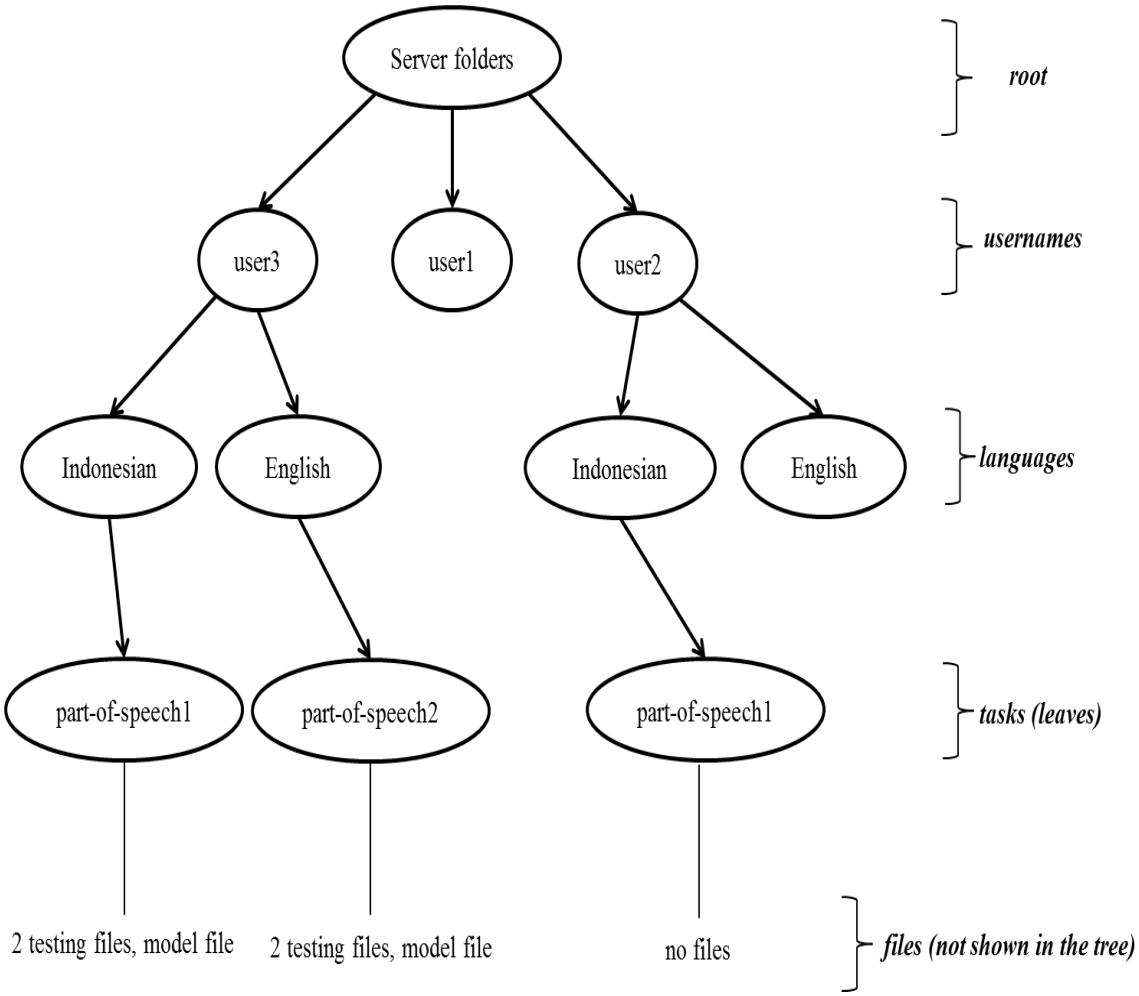

Figure 4. *Example of the tree structure for the folders on the server*

The algorithm shown in Program 1 takes into account all of the above restrictions and problems. It goes through the four levels of folders and even though the last level of folders is not shown to the users it is still useful for better organization. The algorithm checks if the needed files, whose number dependents on the particular stage, are present inside. If so, the three parent folders (username, language, and task) will be included in the tree. Furthermore, the algorithm only looks into the folders on the fourth level which are relevant for the particular stage. For instance, the testing folder is not connected to the training stage so it and the files inside will not be considered.

The code in Program 1 seems a bit complicated because it also needs to handle some special cases. Sometimes it is necessary to exclude whole branches (from one of the usernames down to a task folder) if they do not have any files that fit the requirements (for example, the whole branch for user2 in Figure 4). On occasion, the whole tree

```
foreach (var user_directory in directoryInfo.GetDirectories())
//user directories
{
   foreach (var lang_directory in user_directory.GetDirectories())
   //language directories
   {
      foreach (var task_directory in lang_directory.GetDirectories())
      //task directories
      {
        foreach (var directory in task_directory.GetDirectories())
        //directories under the tasks
        {
          if (directory.Name == required_folder)
          //find the required folder
          {
             FileInfo[] files_in_dir = directory.GetFiles();
             CheckSuitableNodes(user_directory, lang_directory,
                task_directory, ref user_text, ref users,
                ref languages, ref languages_text, directoryNode,
                ref files_in_dir, required_folder);
           }
        }
      }
  }
}
```

Program 1. *The first part of the code that populates the tree for the folders*

might not contain any files and then none of the branches satisfy the requirements. Because of that, the whole tree needs to be empty.

Another problem, which appeared from the iterative nature of the algorithm and how it was constructed, was that it would always consider the same username or language folders as different. As a result there would be several copies of the same username, for example user1, as different nodes in the tree, which is, of course, wrong. That is the algorithm had to include some kind of temporary memoization for already added folders so they will not be duplicated. The code shown in Program 2 solved the problem with the same node appearing multiple times, by making sure that each node does not already exist in the tree before it is added to it.

```
TreeNode user_directoryNode = new TreeNode(user_directory.Name);
TreeNode lang_directoryNode = new TreeNode(lang_directory.Name);
TreeNode task_directoryNode = new TreeNode(task_directory.Name);

//if the username hasn't appeared previously
if(!user_text.Contains(user_directoryNode.Text))
{
   user_text.Add(user_directoryNode.Text);
   users.Add(user_directoryNode);
   directoryNode.ChildNodes.Add(user_directoryNode);
   //if language appears for the first time
   if (!languages_text.Contains(lang_directoryNode.Text))
   {
      languages_text.Add(lang_directoryNode.Text);
      languages.Add(lang_directoryNode);
      user_directoryNode.ChildNodes.Add(lang_directoryNode);
      lang_directoryNode.ChildNodes.Add(task_directoryNode);
   }
   else//if the language was already seen
   {
      int temp_index_lang=
         languages_text.IndexOf(lang_directoryNode.Text);
      TreeNode temp_node_lang=(TreeNode)languages[temp_index_lang];
      temp_node_lang.ChildNodes.Add(task_directoryNode);
   }
}
else//if the username has appeared previously
{
    int temp_index = user_text.IndexOf(user_directoryNode.Text);
    TreeNode temp_node = (TreeNode)users[temp_index];
    //if language appears for the first time
    if (!languages_text.Contains(lang_directoryNode.Text))
    {
       languages_text.Add(lang_directoryNode.Text);
       languages.Add(lang_directoryNode);
       temp_node.ChildNodes.Add(lang_directoryNode);
       lang_directoryNode.ChildNodes.Add(task_directoryNode);
    }
    else// if the language was already seen
    {
       int temp_index_lang =
          languages_text.IndexOf(lang_directoryNode.Text);
       TreeNode temp_node_lang =
          (TreeNode)languages[temp_index_lang];
       temp_node_lang.ChildNodes.Add(task_directoryNode);
    }
 }
```

Program 2. *The second part of the code that populates the tree for the folders*

5.3 Preprocessing of the input files

Preprocessing is the preliminary stage and the place from where one starts whenever a new model needs to be made. The first thing that the users need to do is to decide if the optional tag validation will be done. If so, this needs a file that contains all the tags which are stored by the software for later verification. Whether tag validation is done or not, the next step is to upload the actual input files. These files contain the material from which the model learns how to do the part-of-speech tagging and the test data at the same time. In this case, they always contain at least the word (or punctuations like commas, semicolons, and periods), an underscore, and two tags on each line.

Let us consider an example, in Indonesian and without any noisy data, from one of the input files that were used:

1	Untuk	_	ADP	ADP
2	mengembangkan	_	VERB	VERB
3	batik	_	NOUN	NOUN
4	Jombang	_	NOUN	NOUN
5	,	_	.	.
6	berbagai	_	ADJ	ADJ
7	usaha	_	NOUN	NOUN
8	dilakukan	_	VERB	VERB
9	oleh	_	ADP	ADP
10	Ibu	_	NOUN	NOUN
11	Hj	_	NOUN	NOUN
12	.	_	.	.
13	Maniati	_	NOUN	NOUN
14	.	_	.	.

The words form a sentence when read in vertical order. The input files must have this format for any language and the information contained within must be correct. The actual number of input files is irrelevant as seen in Figure 5.

Once the files are uploaded, if the user chose to do the tag validation the software will check if all of them contain only the tags that were stored previously. If any of the files do not conform to this rule, then they are rejected and will not be considered in the process at all. The software then creates a temporary file, opens a stream for writing to it and goes through all the input files. The needed information is extracted from the files and all of it is stored into the temporary file.

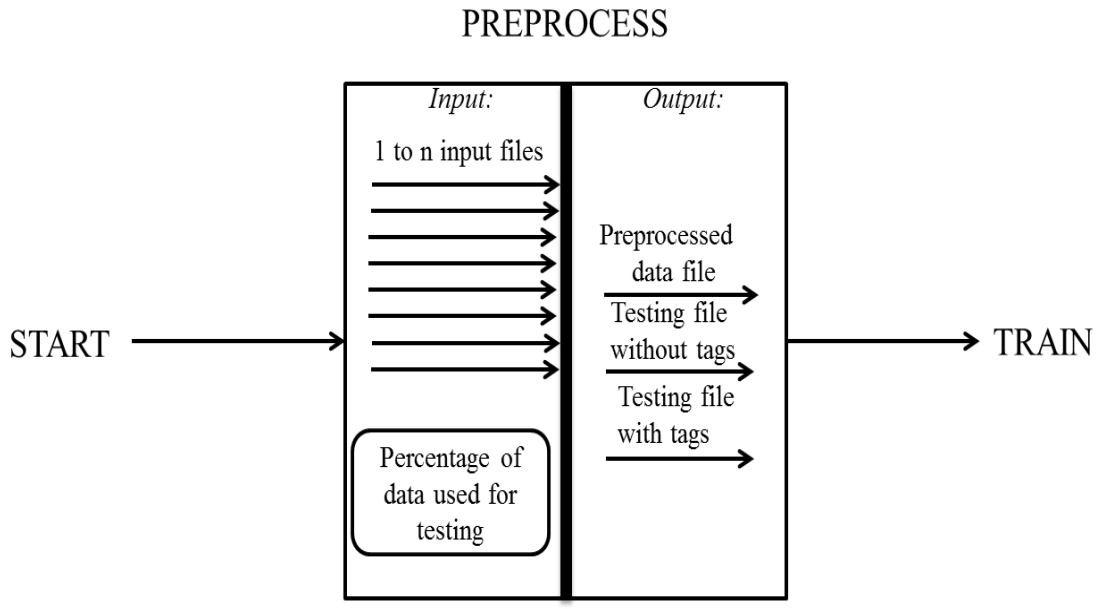

Figure 5. *Overview of data preprocessing*

Then the users are required to enter a percentage (Figure 5) that will represent how much of the data from the temporary file will be for testing purposes; the rest will be for training. The code calculates the number of sentences that will go into the file for testing, based on the percentage, and extracts them in a random manner and places them in a file called test file with tags. For example, if 20% of the data is chosen for testing, out of 1000 sentences, 200 random sentences will go into the test file. Afterwards, each of the test sentences is also stripped of its tags and copied to another file, called test file without tags. Both of those files will later be used in the testing stage, one to be marked by the tagger and the other to validate the results. The sentences that remain in the temporary file are transferred to the file that will be used in the training stage, called pre-

processed data file. Once the three output files are created (Figure 5), the temporary file is deleted since it is no longer needed.

5.4 Training a part-of-speech model

Training is the stage of the process where, the software will try to use the OpenNLP engine to create a model from one of the output files from the previous stage, the pre-processed data file. For a visual representation of this stage see Figure 6. The main functionality for this stage is mostly written in Java since its purpose is to invoke the functionality from the engine.

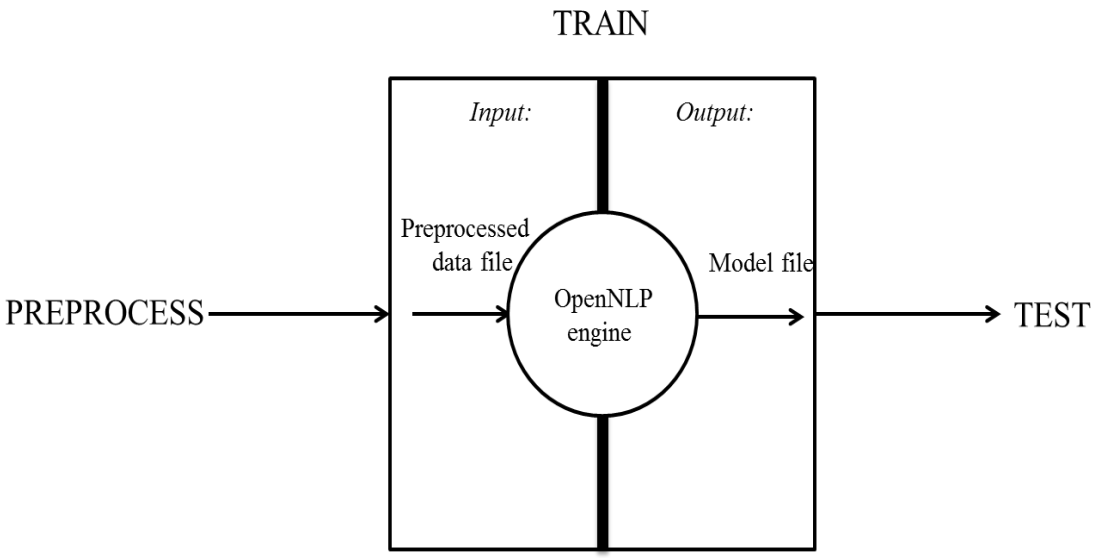

Figure 6. *Overview of training*

The contents of the preprocessed data file are then read and passed to OpenNLP. Other elements that are passed to the engine include type of model to create, that maximum entropy needs to be used, that the task is part-of-speech, and factory parameters that are needed for the procedure [1][2]. Their use can be observed in Program 3. After this, all the data needs to be written to the model file, which is the output from this stage.

```
//read the training data
dataIn = new java.io.FileInputStream(path_to_train_data);

//open the needed streams
ObjectStream lineStream = new PlainTextByLineStream(dataIn, "UTF-8");
ObjectStream sampleStream = new WordTagSampleStream(lineStream);

//initialize the required parameters for the model
TrainingParameters trainParams = new TrainingParameters();
trainParams.put("model", ModelType.MAXENT.name());

//train the model
model = POSTaggerME.train("id", sampleStream, trainParams, null, null);

//write the data to the model file
OutputStream modelOut = null;
modelOut= new BufferedOutputStream(new FileOutputStream(path_to_model));
model.serialize(modelOut);
```

Program 3. *A part of the code of the function that trains the model*

5.5 Testing a model

Testing examines how the model, created in the previous stage, fares when used on unannotated data. There are three input files used here from the preceding stages: test files with and without tags, from the preprocessing stage, plus the model file from the training stage.

By loading the model, the process is able to create a part-of-speech tagger that contains all the knowledge contained within. The tagger will be doing the central job in this stage, which is the identification of the part-of-speech tags for the words, when the data from the test file without tags is given to it.

The prediction is done sentence-by-sentence, while the validation word-by-word. The test file with tags is used to check the validity of the annotation and to calculate the precision of the tagger. This is done by comparing how each word is tagged in sentences from the two test files (Program 4). If the two tags are the same then the tagger was cor-

```
while ((line_no_tags = read_no_tags.ReadLine()) != null
&&(line_with_tags = read_with_tags.ReadLine()) != null)
//read until the EOF
{
   string predictTags = tagger.tag(line_no_tags);
   //predict the tags for a sentence
   predictTags = predictTags.Replace("/", "_");
   //replace the tag marker from OpenNLP with ours
   string[] wordsInPredictTags=predictTags.Split(' ','\t','\n');
   //split the line into words
   string[] wordsInFileWithTags=
   line_with_tags.Split(' ','\t','\n');

   for(int i = 0; i < wordsInPredictTags.Length; i++)
   {//output the results to a file
      write_log.Write(wordsInPredictTags.ElementAt(i)+"========="
        +wordsInFileWithTags.ElementAt(i)+"\r\n");

      //check the validity of the predicted tags
      if (wordsInPredictTags.ElementAt(i).Equals
         (wordsInFileWithTags.ElementAt(i)))
      {//if the tag is correct
         wordAccuracy.add(1);//add 1 to the accuracy
      }
      else
      {//if the tag is not correct
         wordAccuracy.add(0);//add 0 to the accuracy
      }
   }
}
```

Program 4. *A part of the code of the function that tests the model*

rect (which adds one to the accuracy), if they are not equal then the tagger made a mistake (that adds zero to the accuracy). Then the precision is calculated by dividing the accuracy by the number of words that were tagged. The outputs from this stage of the process are the precision of the tagger and a log files where the users are able to get the results and to compare all the right and wrong tags (Figure 7).

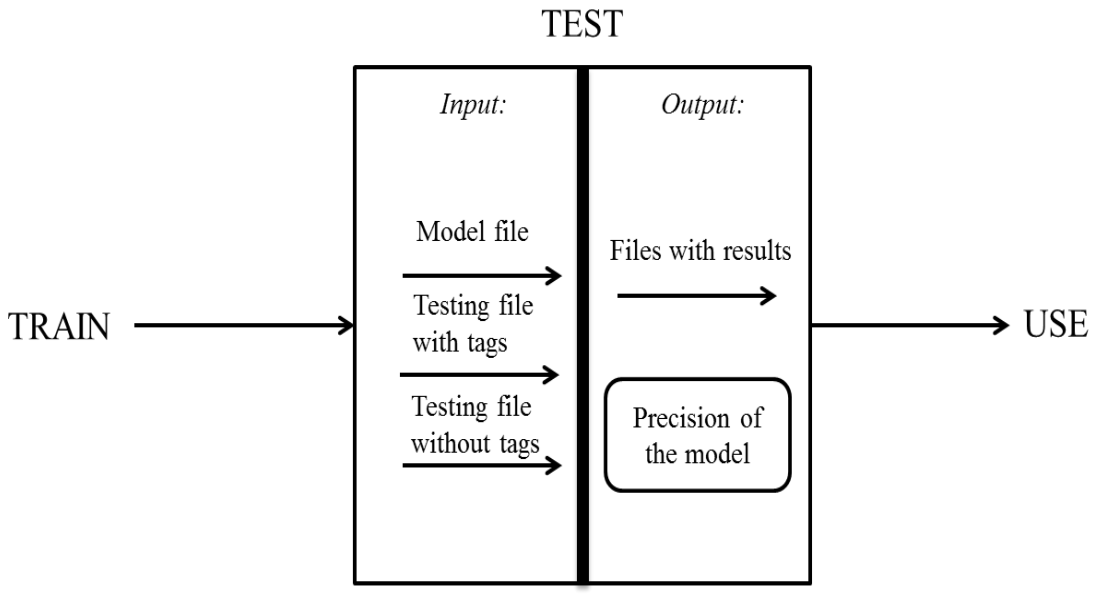

Figure 7. *Overview of testing*

5.6 Using a model

In a production run the model will be used on some new data since the testing stage has been satisfactorily completed. This data can be anything: random sentences from the internet, online articles, or eBooks, as long as the users gather all that data into one file. Its name can be either extensionless or with the .txt extension.

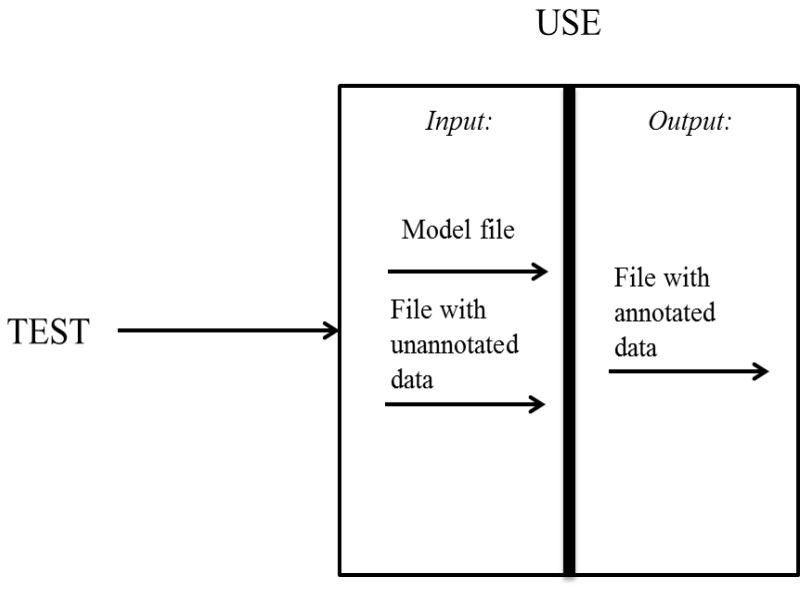

Figure 8. *Overview of model use*

Hence, there are two inputs which are used here (Figure 8), the model file and a file with data that is not annotated, which must be in the same language as the input files from the preprocessing stage.

Similarly to testing, the model is loaded into memory, a part-of-speech tagger is initialized based on its knowledge, all the sentences are given to the tagger, word by word, and this predicts the tags for each one of them. The resulting data is written to an output file where the user can observe the results from the annotation of the input data (Figure 8).

This last stage of the process can be especially useful for those users who are very satisfied with the high precision of a model that they have created in the past and do not want to waste time with the other functionality since their goal has been achieved. The production run, like all the stages, can be done countless number of times as long as the required model file is somewhere on the server or on the local machine that is being used.

6. EXPANDABILITY OF THE APPLICATION

One of the main objectives for the software was that it needed to be easy to expand. The expandability needs to be satisfied on several different levels: languages, natural language processing tasks, and engines.

6.1 Language expandability

The first level is the ease to create models for different natural languages. One example of this would be the possibility for the users to create part-of-speech taggers for English, Finnish, German, and other languages.

It is really important that the users themselves find suitable input files. This could be solved with various corpora and Treebanks [36] available on the internet for a large number of languages [7][41]. Moreover, any kind of texts on different topics can be used as inputs. Once the correct files are given to the software and their language is indicated to the application, the rest is handled by the code automatically.

Some further checks can be implemented to make the choices for the languages error-proof. For instance, the software could check if the language in all the input files is the same with the one that has been indicated to the software. This could be done with some language identification algorithm. Another possible method, that could be implemented, would be to extract the language from the files themselves. The language will have to be indicated somehow, for instance, by having the language code in the filename. And yet another method could use the combination of both mentioned above, that would achieve greater automation in the process.

6.2 Task expandability

The second level of expandability is the number of natural language processing tasks. Some additional examples of these would include: named entity recognizer (also known as name finder), parser, and grapheme-to-phoneme convertor. Grapheme-to-phoneme conversion is used to make predictions on how words are pronounced. All of these are recommendations from the users as tasks that could be used in the future to expand the

functionality of the software. Moreover, some of the other tasks offered by OpenNLP might also be used, although this is not the case with OpenNLP parser. Another engine will be used to implement the parsing task.

One way to make the task expandability possible and easy to apply is make use of inheritance, with some of the classes that are already implemented in the code. A base class contains a separate abstract method for every stage of the process: preprocess, train, test, and use. These will represent all the common functionality between all the tasks. Because of that, each task can have a separate class, all of which are derived from the base, and therefore inherit the methods and have distinct implementations for the functions.

All the potential tasks, named entity recognizer, parser, and grapheme-to-phoneme convertor, need a model to be trained and then evaluated. Moreover, depending on the requirements of each task, different forms of preprocessing will need to be implemented. Furthermore, with the many different standards for the structure of the data being used, the need for extensive preprocessing is quite understandable. Additionally, the stage using a model is also needed for the future tasks. Because of these reasons all the classes for the tasks can safely inherit the common methods from the base class and use its original implementations or override them and create new functionality. [1][2][12][31]

This expandability is also supported by the folder structure, which was discussed previously. Since every task has a different name from the others and there are no plans to have two different implementations of the same task, for instance, from different engines, the current structure can still be used even after extensive expansion of the application. Furthermore, some of the potential tasks require part-of-speech tagged data to be used along with some additional analysis. An example of this would be the parsing. Because of that, it represents a good follow up to the already implemented part-of-speech tagger.

6.3 Engine expandability

The third level of expandability is the number of natural language processing engines included. One of the original requirements for the software was to include at least two of these: OpenNLP and SharpNLP [38]. The integration of OpenNLP was achieved, with some challenges along the way, but not so with SharpNLP, although it first appeared that things would be the other way around. This was because OpenNLP is based on the Java programming language while SharpNLP is based on C#, and since the software for this thesis was done with ASP.NET it was also written in C# [38]. The failure to implement the latter engine happened because of the lack of any kind of documentation about its implementation or use and the lack of support, since the project has been

abandoned. Another, smaller problem was that the engine was specifically designed as a Windows form application and here it was necessary to integrate the engine into the natural language processing application [38]. Some further examples of engines that the users would like see integrated in the future would include Maltparser [11] and Phonetisaurus [30].

An approach similar to the one for the tasks can also be used on this level to achieve expandability. A base class can be used that will contain abstract methods and properties, as placeholders, that are common for all the engines. In addition, every new engine will have its own separate class, which will inherit from the base one. Hence, each of the descendant's methods will actually interact with the appropriate engine. Moreover, they can also have their own implementation of the functions that are inherited from the base class.

This structure goes well hand-in-hand with the fact that all the engines are based on diverse programming languages. Maltparser is based on Java [11], while Phonetisaurus is based on Python [30]. That is why every class for each engine will have a code that interacts with them in a different programming language. Because of these reasons, some further libraries and plugins will have to be included and applied, so that these engines can be used in the .NET framework and in the application. But with the likes of IKVM.NET [9], and IronPython [16], it is very probable that the implementation of the engines can be achieved.

Of course, it should be noted that there is a connection between the structure of classes for the engines and the structure of classes for the tasks, as seen in Figure 9. Their relationship, in this case, is composition, that is, every engine owns its tasks. A concrete example of this would be the OpenNLP and its part-of-speech tagging implemented.

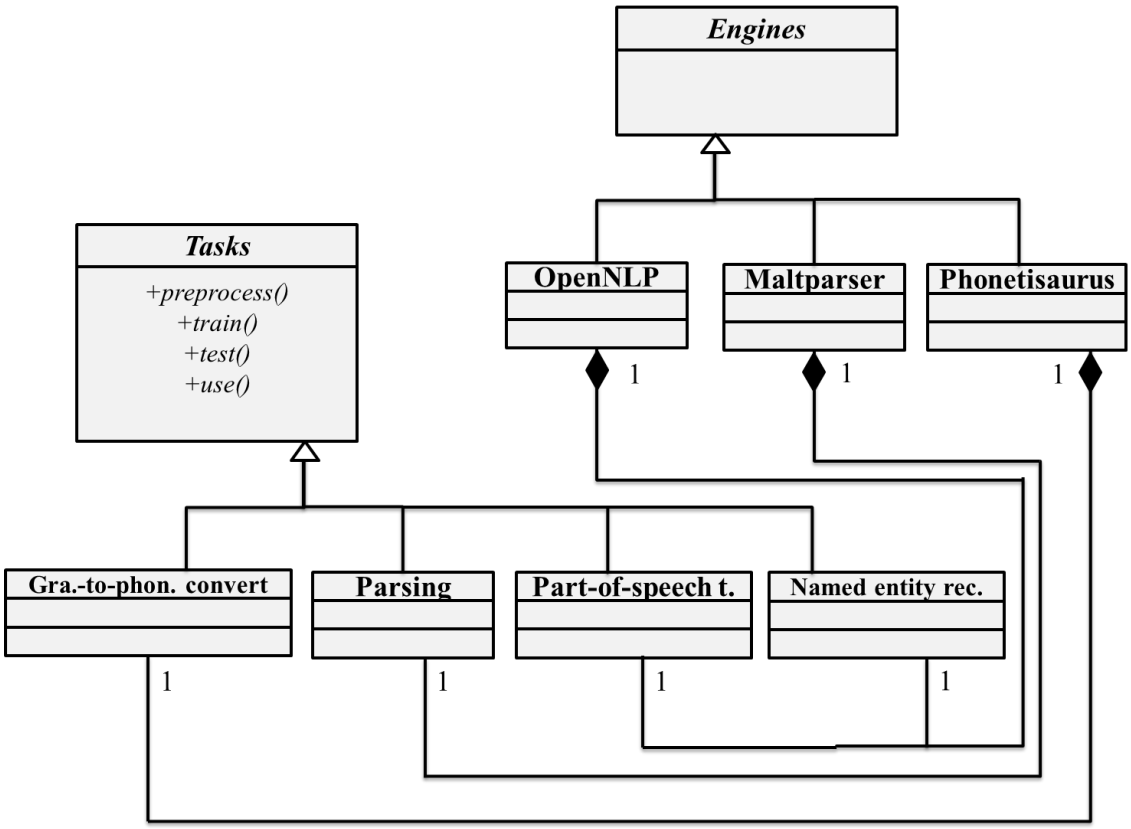

Figure 9. *Classes for tasks and engines*

7. EVALUATION

Let us take a look at a short summary (in Table 1) of the criteria and how they were fulfilled during the development.

Table 1. *Criteria and their fulfillment*

Criteria	Fulfillment
No technical prerequisite knowledge needed to use the application	The application handles everything automatically while the process is followed
Flexibility should be allowed in the workflow	Multiple entry and exit points in the process; continuing/pausing of process runs; data is automatically saved
The application should be expandable with new tasks and engines	New tasks and engines can be added to the application; supported by both the folder and system structure
The software should be portable	The interface is a web application and consequently is portable
The tasks and engines should be accessible from various locations	The interface is a web application and, therefore, can be accessed from different locations

Therefore, it can be concluded that all the criteria were taken into consideration and were used as guidelines while the application was developed.

Effects of the constraints

We also faced some constraints while developing the application, all of which were external. When facing the constraints it was noticed that some of them brought effects to the system, several of which had a positive impact on the work while others not so much. A number of the effects that are discussed below might also be left for the future.

One of the effects that could appear from the restriction of using ASP.NET is that the application might only be available for use in the Microsoft development environment. This includes operating systems, servers, etc.

Two effects appeared from the implementation of OpenNLP. The first one was the difficulty of integrating it into the application since the engine was written in Java, that is, until IKVM.NET [9] was finally found. This allowed implementing and runing Java into ASP.NET back-to-back with C# [9], thus simplifying this project, thankfully. Because of IKVM it was possible to include the OpenNLP API code in the software and to do the functions that accessed the functionality of the engine [9].

The second effect is that once a model is trained and created from the input files no other new data can be added to it. For instance, if a part-of-speech tagger is created for English, then if it is trained and tested with some data, there is no possibility to retrain it with some other data. One would have to gather the input files that were used in the first place, add the new input files and train a completely new model. It should be noted that the users did not find this much of a problem and, therefore, this issue was left as it is. [1][2]

At the same time, the implementation of engines should not be stopped with OpenNLP. The interface should also be expandable. A side effect from the expandability of the engines is that the code may become too dependent on and cluttered by the various libraries and plugins that might need to be used in the future. This comes from the fact that most of the potential engines are based on various programming languages that, by default, are not native with the framework that is used.

Another constraint was that the different files used should not be checked whether they are valid. This was required to be able to focus more on the implementation of the OpenNLP engine and its part-of-speech task. Moreover, the input files that will be used by the users are based on corpus files which have standard structure and are likely to be error free. So, presuming that the inputs are correct, all the other files created in the application should be correct by extension. Even though there is no validation of the integrity of the files that are used in the application, there are numerous error handlers to deal with any kind of mistakes that the users might make or that may happen during execution.

Another effect, which comes out from how the whole process of creating the part-of-speech tagger is laid out, is that the redoing of some of the stages is not possible. Because of that, the re-preprocessing of some additional data after the preprocessing is finished cannot be done. For now, the process must be restarted and the new data needs to be included in the input files, after which the preprocessing should be done. A similar approach needs also to be taken for the testing stage. In the current state of the application, the only truly reusable stage is using a model. That is the only stage through which the users can make multiple runs without any need to restart the process.

8. SUMMARY

At the end, the resulting system allows the task of part-of-speech tagging using the OpenNLP engine to be used without much difficulty by non-technical users that have background in linguistics, for example. By going through the four stages of the process the users are able to prepare their data, use the processed data to create a tagger, evaluate the part-of-speech model, and finally use the tagger according to their needs.

Additionally, the process in the application includes multiple entry and exit points. The users are free to enter the creation flow at any of the four stages and use them. Their usage is, of course, dependent on the requirements for every one of stages. That flexibility is also supported by the automated storage of the data from each stage in case the users want to quit the application and continue from where they stopped, later on.

At the same time, the application should be expandable with new tasks and new engines. Some of the would-be additions to the software would require additional libraries or plugins to be used, but, nevertheless, their implementation should not be harder than the one for the OpenNLP engine and its part-of-speech task. Additionally, the current state of the whole system also allows the creation of natural language task models for different languages. Moreover, since the OpenNLP engine is already implemented and in use, the application can easily be extended with any of its tasks, in a fashion similar to the part-of-speech tagging.

Moreover, the application is also accessible and portable. Through the use of the Microsoft ASP.NET framework the system was created in the form of a web application. Hence, the software can be used by any user that has access to the server without the need to install anything on their machine except for a web browser.

Future work

Some further modifications could be made to particular stages of the process in order to make them more reusable. The preprocessing could be improved by adding the possibility to re-preprocess some additional data. For example, if one forgets to add all the input files for preprocessing it should be possible to add them even when that stage is almost at its end. This could be achieved by splitting the new data based on the same percentage used previously and appending the data into the testing and training files, respectively. Using this method the testing against training data ratio will still be preserved and they will be attached correctly.

The testing stage can also be somewhat improved by adding the feature of re-testability. This could be achieved by allowing the users to upload some new annotated data after which the software could create two new test files from it. They would replace the test files from the previous run. Subsequently, the model could be re-tested using the two new files.

The use of a model should also be changed so that more than one unannotated data file could be used by the model. The software could create one file with the data from the various inputs and pass that file to the tagger for marking. If this is implemented in the future the users could upload any number of files to be annotated by the model.

Other additions or improvements would include better algorithm for populating the tree for the folder structure, various checks of the different files that are created from the process, reading of any type of files as input, and renaming of files and folders by the users. The algorithm for the folder structure can also be improved by increasing its efficiency. One way to enhance this would be to remove the dependency of storing nodes so that their existence can be checked in the tree. By using some better memoization techniques or by implementing a better database the mentioned objectives could be achieved.

Another improvement that could be made to the system in the future would be the inclusion of better checks to handle the input and output files and the inclusion of a database for file validation results. For example, integrity tests could be placed to make sure that all the files have the correct format. It could even be checked if those are the needed files for the specific stage of the process and store that information in a database. This would lead to better error handling and that would lead to fewer mistakes made by the users.

The application could also be made to read various types of files instead of the selected few that are used now. This way it would be possible to use some files that are currently not supported, say, Word or Excel files. For now, they are not supported because they require different methods to be read.

An enhancement to the folder and file structure could also be made. For instance, one possibility could be to allow the users to rename the directories and files that are created in the application. This might lead to their better organization on the server but it also may create problems since everything on it is shared by all the users. This way one user could rename the data of another by mistake. And that would lead to the need of implementation of various security measures like: file and folder restrictions, permission to use/rename data, and rights for the users. Another way would be to implement a database to allow all of the above plus sorting and viewing of data in various ways.

REFERENCES

[1] Apache OpenNLP Developer Documentation, The Apache Software Foundation, website. Available (accessed on 15.10.2014): http://opennlp.apache.org/documentation/1.5.3/manual/opennlp.html

[2] Apache OpenNLP Tools 1.5.3 API, The Apache Software Foundation, website. Available (accessed on 19.10.2014): http://opennlp.apache.org/documentation/1.5.3/apidocs/opennlp-tools/

[3] A. L. Berger, V. J. Della Pietra, S. A. Della Pietra, A maximum entropy approach to natural language processing, Computational Linguistics, Volume 22, Issue 1, 1996, pp.2–13.

[4] A. E. Borthwick, A Maximum Entropy Approach to Named Entity Recognition, New York University, 1999, pp.18–25.

[5] E. Cambria, B. White, Jumping NLP Curves: A Review of Natural Language Processing Research, IEEE, 2014, pp. 3–5.

[6] A. Chopra, A. Prashar, C. Sain, Natural Language Processing, International Journal of Technology Enhancements and Emerging Engineering Research, Vol.1, No.4, 2013, pp.131–133

[7] Corpora and other language and speech data under DICE, The University of Edinburgh, website. Available (accessed on 5.11.2014): http://www.inf.ed.ac.uk/resources/corpora/

[8] M. Crisan, Chaos and Natural. Language Processing, Acta Polytechnica Hungarica, Vol. 4, No.3, 2007, pp. 61–62.

[9] J. Frijters, IKVM.NET Home Page, website. Available (accessed on 2.10.2014): http://www.ikvm.net/

[10] B. Habert, G. Adda, M. Adda-Decker, P. Boula de Mar¨euil, S. Ferrari, O. Ferret, G. Illouz, P. Paroubek, Towards Tokenization Evaluation, First International Conference on Language Resources and Evaluation (LREC), 1998, pp. 1–2.

[11] J. Hall, J. Nilsson, J. Nivre, MaltParser, website. Available (accessed on 2.11.2014): http://www.maltparser.org

[12] J. Hall, J. Nilsson, J. Nivre, MaltParser User Guide, website. Available (accessed on 2.11.2014): http://www.maltparser.org/userguide.html

[13] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques Third Edition, Morgan Kaufmann, 2012, pp. 6–124.

[14] R. C. Hill, R. C. Campbell, Maximum Entropy Estimation in Economic Models with Linear Inequality Restrictions, Departmental Working Papers 2001–11, Louisiana State University, pp.1–2.

[15] J. Hockenmaier, G. Bierner, J. Baldridge, Extending the coverage of a CCG System, Research in Language and Computation 2, 2004, pp.203–204

[16] IronPython, IronPython Community, website. Available (accessed on 4.11.2014): http://ironpython.net/

[17] E. T. Jaynes, Information Theory and Statistical Mechanics, Physical Review, vol. 106, no. 4, 1957, pp. 620–624.

[18] K. S. Jones, Natural Language Processing: A Historical Review, University of Cambridge, 2001, pp.2–10.

[19] D. Jurafsky, J. H. Martin, Speech and Language Processing: An introduction to speech recognition, computational linguistics and natural language processing, Prentice Hall PTR, 2006, chapter 5, pp.1–15.

[20] D. Klein, C.D. Manning, Accurate Unlexicalized Parsing, ACL '03 Proceedings of the 41st Annual Meeting on Association for Computational Linguistics, vol. 1, 2003, pp.423–424.

[21] S. B. Kotsiantis, Supervised Machine Learning: A Review of Classification Techniques, University of Peloponnese, 2007, pp. 2–4.

[22] E. D. Liddy, Natural Language Processing, Encyclopedia of Library and Information Science 2nd Ed., Marcel Decker Inc., 2001, pp. 1–10.

[23] R. Malouf, A comparison of algorithms for maximum entropy parameter estimation, COLING-02 proceedings of the 6th conference on Natural language learning, Volume 20, 2002, pp. 1–6.

[24] C. D. Manning, Part-of-Speech Tagging from 97% to 100%: Is It Time for Some Linguistics?, Computational Linguistics and Intelligent Text Processing, 12th International Conference, CICLing 2011, Proceedings, Part I, 2011, pp.171–173.

[25] C. D. Manning, H. Schütze, Foundations of Statistical Natural Language Processing, The MIT Press, 1999, pp. 589–593.

[26] L. Marquez i Villodre, Part-of-speech Tagging : A Machine Learning Approach based on Decision Trees, Universitat Politecnica de Catalunya, 1999, pp. 16–20.

[27] Maximum Entropy Framework, SourceForge, website. Available (accessed on 5.10.2014): http://maxent.sourceforge.net/about.html

[28] I. J. Myung, M. A. Pitt, Applying Occam's razor in modeling cognition: A Bayesian approach, Psychonomic Bulletin & Review, Volume 4, Issue 1, 1997, pp.79–80.

[29] T. Naseem, B. Snyder, J. Eisenstein, R. Barzilay, Multilingual Part-of-Speech Tagging: Two Unsupervised Approaches, Journal of Artificial Intelligence Research 36, 2009, pp.1–3.

[30] J. R. Novak, Phonetisaurus, website. Available (accessed on 2.11.2014): https://code.google.com/p/phonetisaurus/

[31] J. R. Novak, Phonetisaurus ReadMe, website. Available (accessed on 2.11.2014): https://code.google.com/p/phonetisaurus/wiki/ReadMe

[32] OpenNLP Proposal, The Apache Software Foundation, website. Available (accessed on 10.10.2014): http://wiki.apache.org/incubator/OpenNLPProposal

[33] L. Padró, POS Tagging Using Relaxation Labelling, Proceedings of 16th international conference on computational linguistics Coling, 1996, pp.879–880

[34] S. J. Phillips, M. Dudík, R. E. Schapire, A maximum entropy approach to species distribution modelling, ICML '04 Proceedings of the twenty-first international conference on Machine learning, 2004 pp.83–84.

[35] A. Ratnaparkhi, Maximum entropy models for natural language ambiguity resolution, University of Pennsylvania, 1998, pp. 6–51.

[36] D. Rusu, L. Dali, B. Fortuna, M. Grobelnik, D. Mladenic, Triplet extraction from sentences, Proceedings of the 10th International Multiconference Information Society–IS, 2007, pp.8–9.

[37] L. J. Savage, The Foundations of Statistics Reconsidered, Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1, 1961, pp.575–577.

[38] SharpNLP - open source natural language processing tools, Microsoft, website. Available (accessed on 23.9.2014): https://sharpnlp.codeplex.com/

[39] J. Shore, R. Johnson, Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy, IEEE Transactions on Information Theory, vol.26, no.1, 1980, pp.26–27.

[40] R. Socher, J. Bauer, C.D. Manning, A.Y. Ng, Parsing with Compositional Vector Grammars, In Proceedings of ACL, 2013, pp. 455–456.

[41] Statistical natural language processing and corpus-based computational linguistics: An annotated list of resources, The Stanford NLP Group, website. Available (accessed on 5.11.2014): http://www-nlp.stanford.edu/links/statnlp.html

[42] A. Voutilainen, Part-of-Speech Tagging, The Oxford Handbook of Computational Linguistics, 2005, pp.219–227

[43] J. J. Webster, C. Kit, Tokenization as the initial phase in NLP, COLING '92 Proceedings of the 14th conference on Computational linguistics, volume 4, 1992, pp. 1106–1108.